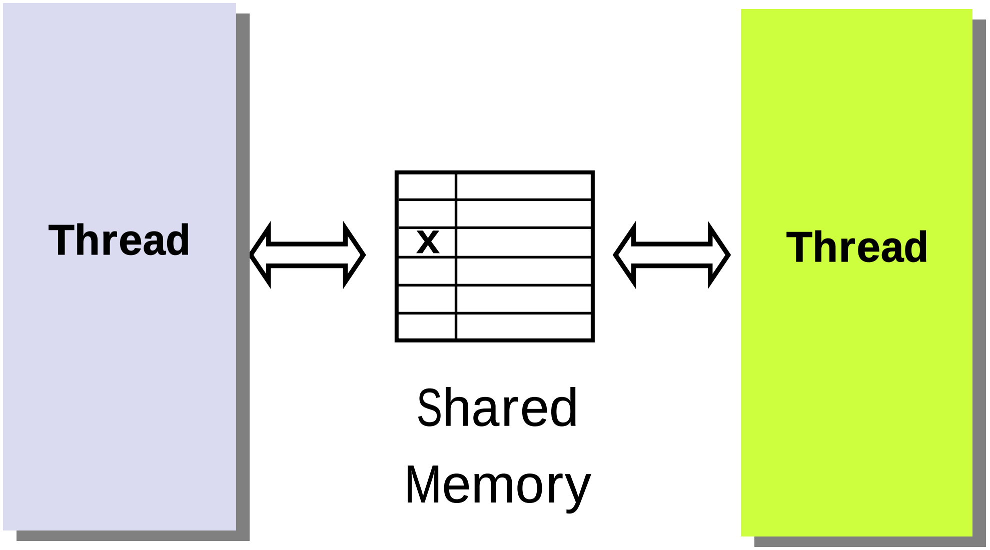
Multithreaded Verification by Context Inference

Tom Henzinger Ranjit Jhala Rupak Majumdar

EPFL UCSD UCLA

Multithreaded Programs



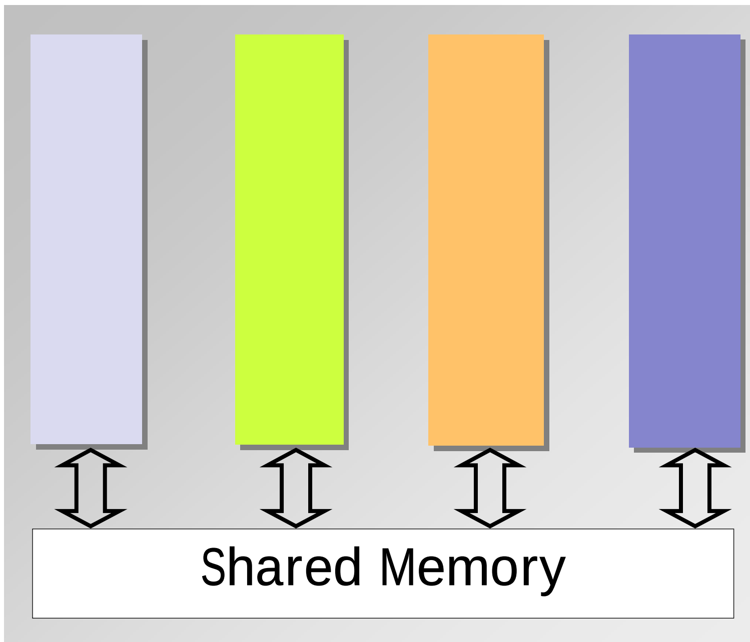
OS, WebServers, Databases, Embedded Systems

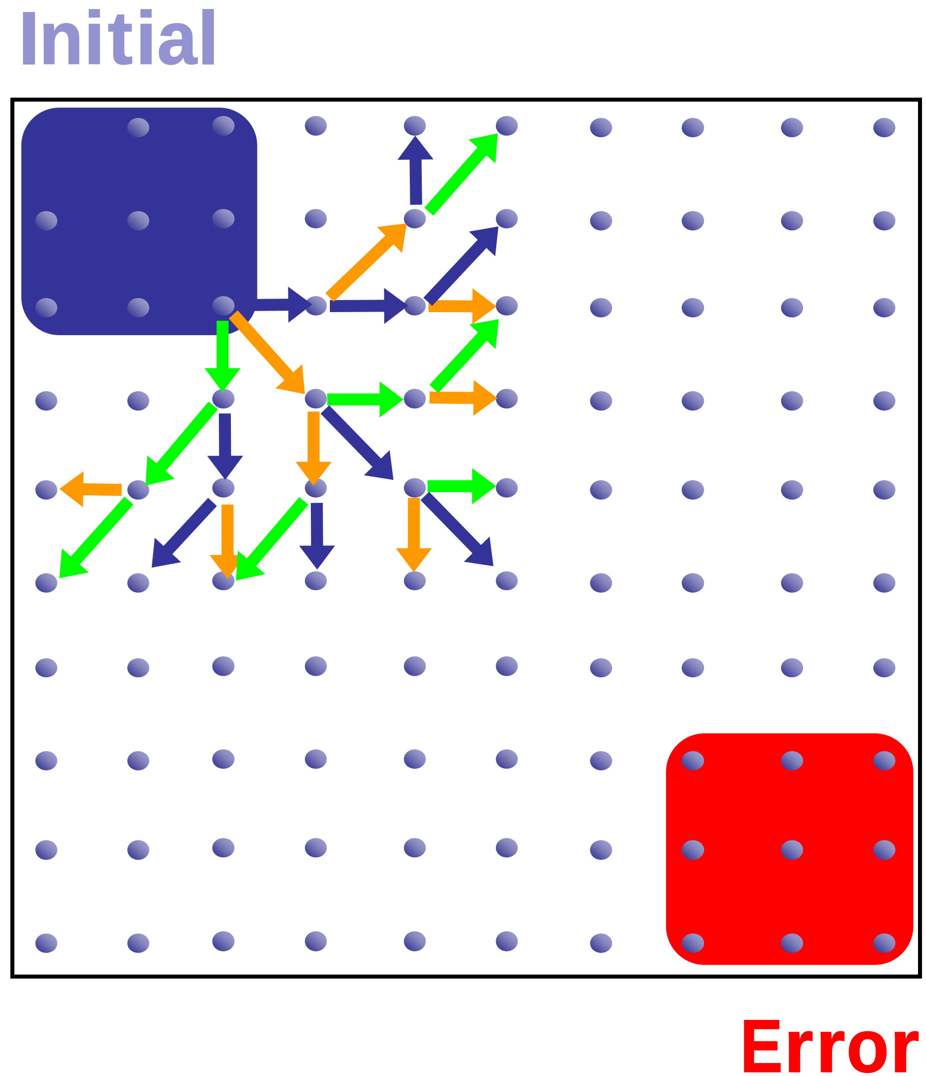
Curse of **Interleaving**

- Non-deterministic scheduling
- Exponentially many behaviors: hard to detect, reproduce errors

Testing exercises a fraction of possible behaviors

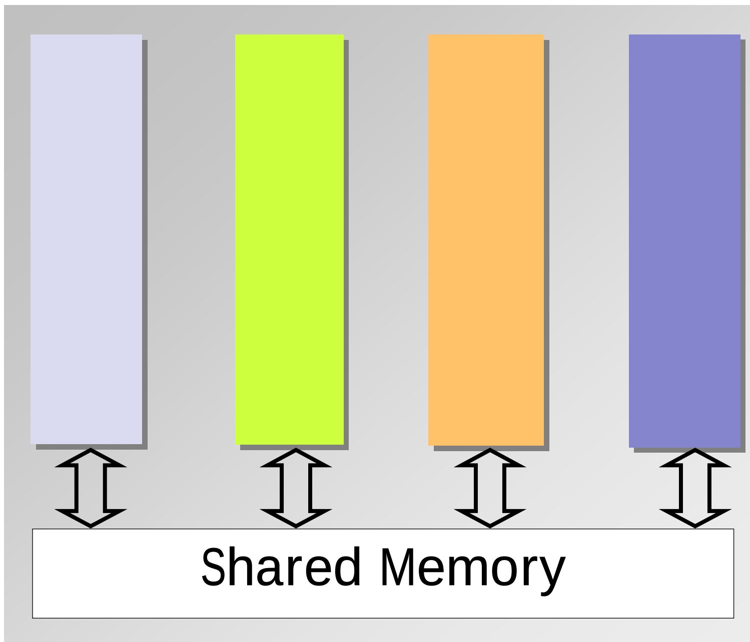
Safety Verification by State Exploration

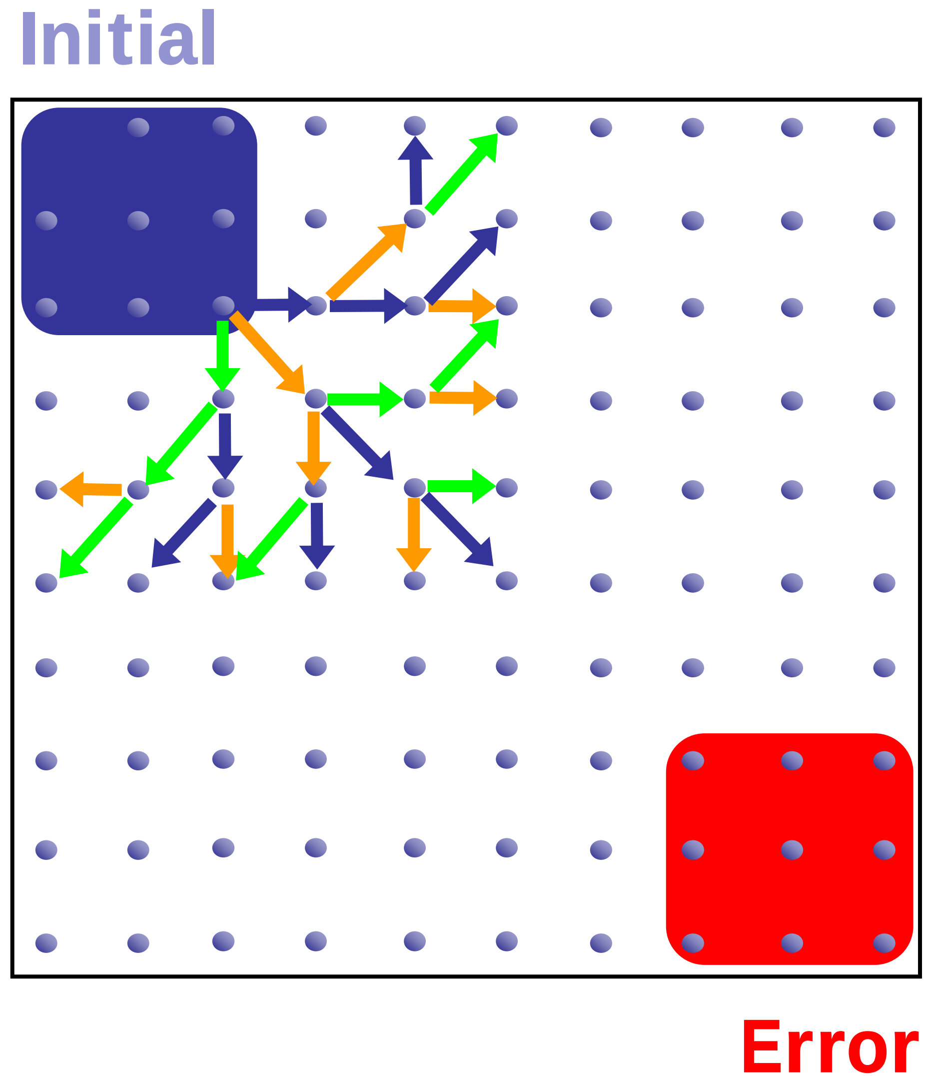




Is there a **path** from **Initial** to **Error** ?

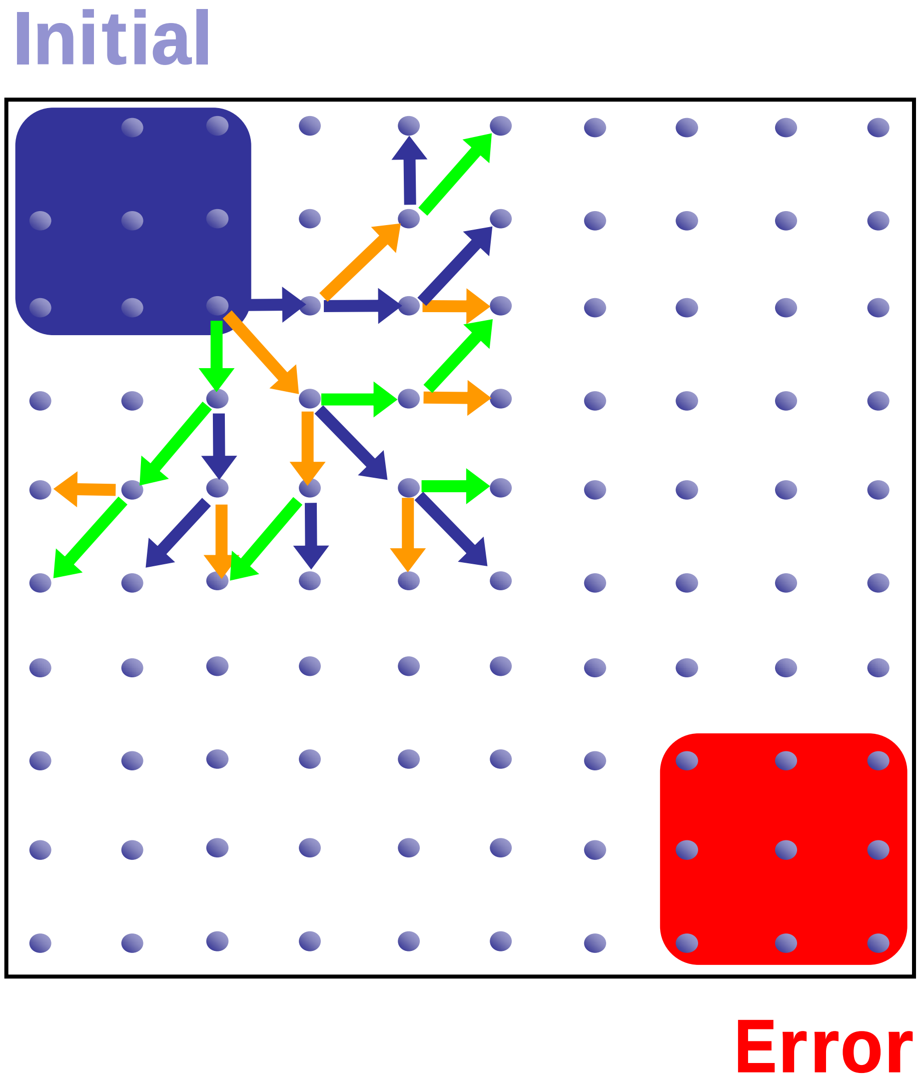
Problem: State Explosion





Is there a **path** from **Initial** to **Error** ?

Problem: State Explosion



1. Data

Infinitely many valuations for program variables

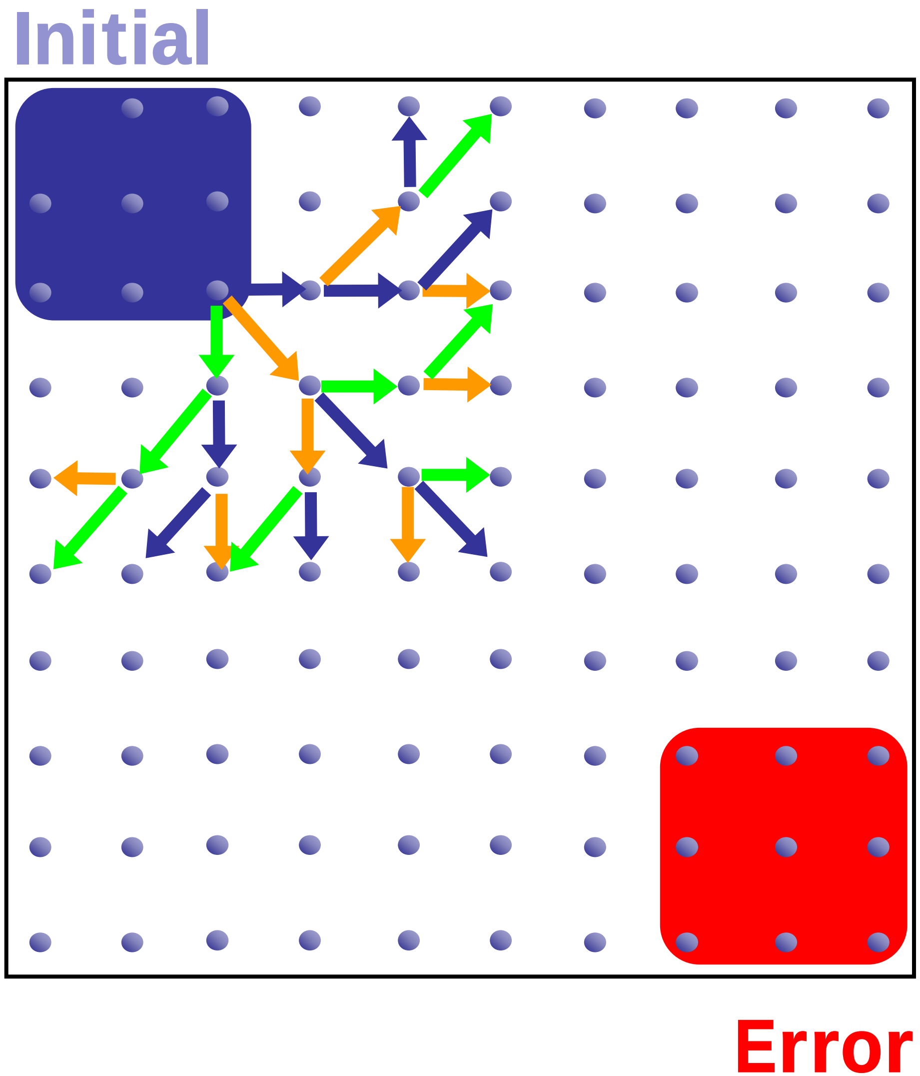
2. Control

k threads, m locations = m^k

- k=4,m=100, states = 1 billion

Unbounded threads ?

Problem: State Explosion



1. Data

Infinitely many valuations for program variables

2. Control

k threads, m locations = m^k

- k=4,m=100, states = 1 billion

Unbounded threads ?

Solution: Abstract Irrelevant Detail

1. Data

Infinitely many valuations for program variables

Observation:

- Few relevant variables, **relationships**
- Track **predicates** (relationships) instead of values

1. Predicate Abstraction

2. Control

k threads, m locations = m^k

- k=4, m=100, states = 1 billion

Unbounded threads ?

Observation:

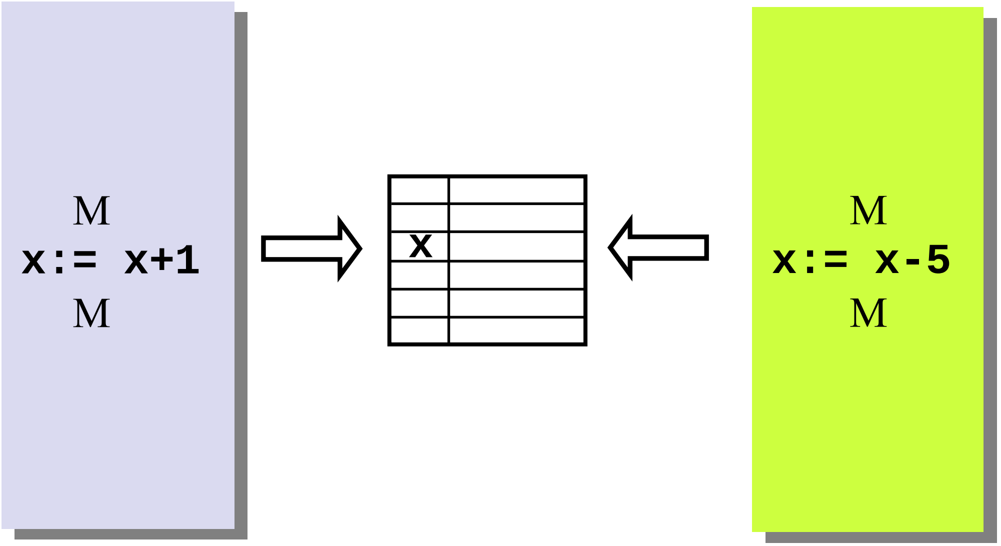
- Analyze system as **Thread + Context**
- Context: Abstraction of **other** threads **(w.r.t. property)**

2. Context Abstraction

Plan

1. **Abstractions against State Explosion**
   - Data : Predicate Abstraction
   - Control : Context Abstraction
2. Thread-Context Reasoning
3. Context Inference
4. Experiments
5. Related Work

Example Property: Data Races



A data race on **x** is a state where:

- **Two** threads can access **x**
- One of the accesses is a **write**

Unpredictable, undesirable

Example Program

```
1: while(1){
     atomic{
2:     old := s;
3:     if(s==0){
4:        s := 1;
       }
   }

 // do_work()
     M

5:   if(old==0){
6:     x++;
7:     s:=0;}
}
```

Check for races on x:

Initially: **s** is 0

1st thread into **atomic** :

- sets **old** to 0 (value of **s**)
- sets **s** to 1
- passes test before access

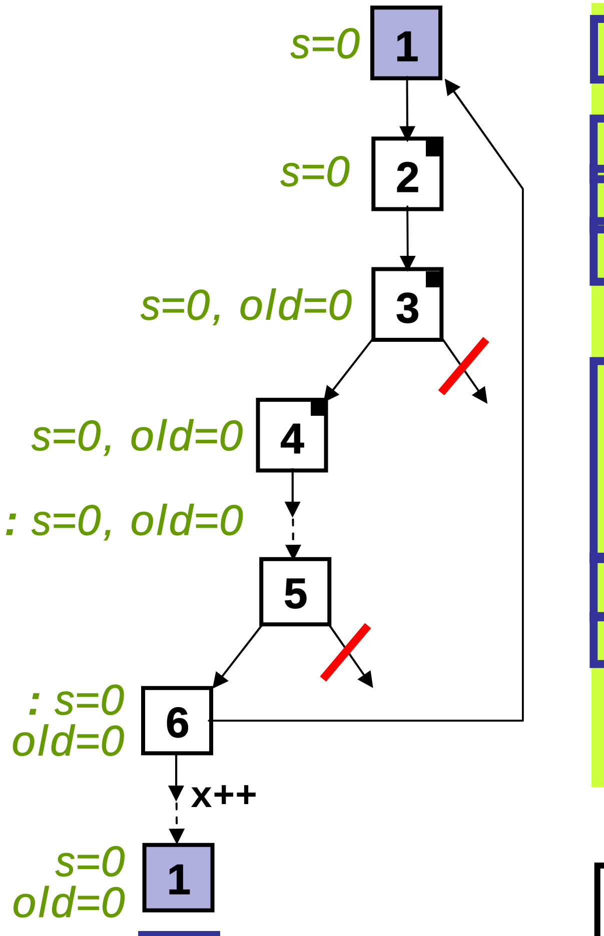
Later threads:

- set **old** to 1 (value set by 1st thread)
- fail test before access (until the 1st thread is done)

Plan

1. Abstractions against State Explosion
   - **Data : Predicate Abstraction**
   - Control : Context Abstraction
2. Thread-Context Reasoning
3. Context Inference
4. Experiments
5. Related Work

Predicate Abstraction



Reachability Graph

```
1:  while(1){
      atomic{
2:      old := s;
3:      if(s==0){
4:        s := 1;
        }
    }

  // do_work()
      M

5:  if(old==0){
6:      x++;
7:      s:=0;}
}
```

Predicates on Variables

s=0, old=0

Q: What about **other threads ?**

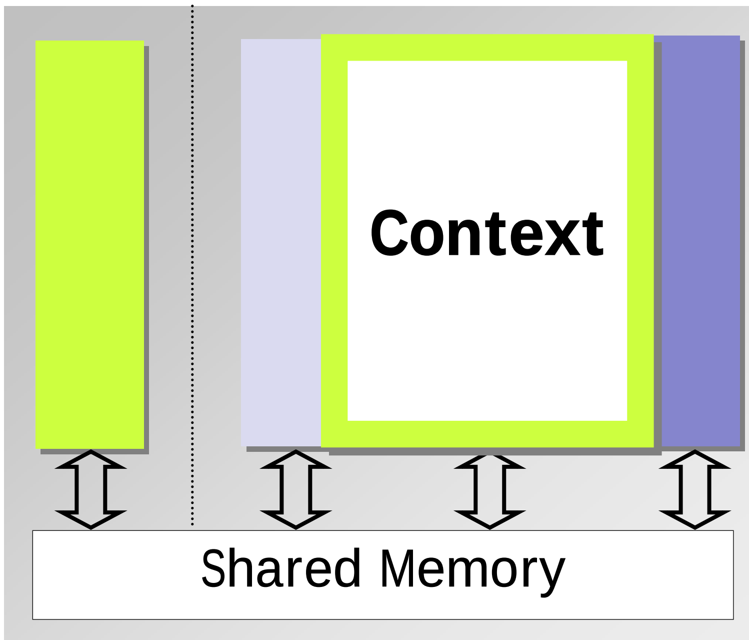
Plan

1. Abstractions against State Explosion
   - **Data : Predicate Abstraction**
   - Control : Context Abstraction
2. Thread-Context Reasoning
3. Context Inference
4. Experiments
5. Related Work

Plan

1. Abstractions against State Explosion
    - Data : Predicate Abstraction
    - **Control : Context Abstraction**
2. Thread-Context Reasoning
3. Context Inference
4. Experiments
5. Related Work

Threads and Contexts



Assume threads run same code

Context:

Summary of all other threads

- Precise enough to check property

System = Thread + Context

Q: What does a **Context** look like?

Q: What about other threads ?

Contexts = Thread Summary + Counting

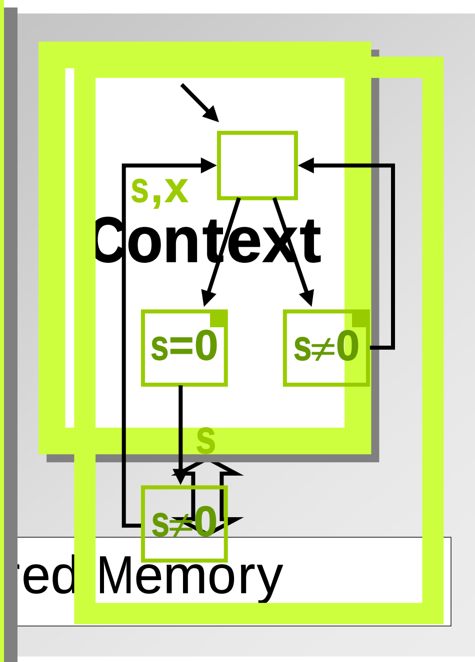
Thread

```
while(1){
  atomic{
   old := s;
   if(s==0){
     s := 1;
   }
  }

 // do_work()
  M

  if(old==0){
    x++;
    s:=0;}
}
```

Summary / **Context**



Context:

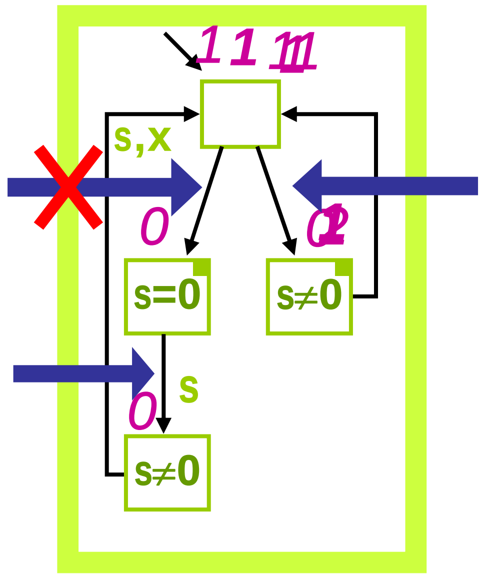
Abstraction of **all other** threads

1. **Summarize** a single thread

2. Multiple threads by **counting**

Multiple Threads by Counting

Context



State

$s \neq 0$ $s=0$

Contra!

Initial loc = ***1*** , other = *0*

Operations

1. Pick **edge** w/ **source counter** > *0*,
2. **Source** counter *-1*
   Target counter *+1*
   Havoc variables on edge,
   Assume predicate on **target**

Unbounded threads

k-Counter Abstraction:

Value > *k* abstracted to ***1***

for *k=1*, values: *0*,*1*,***1***

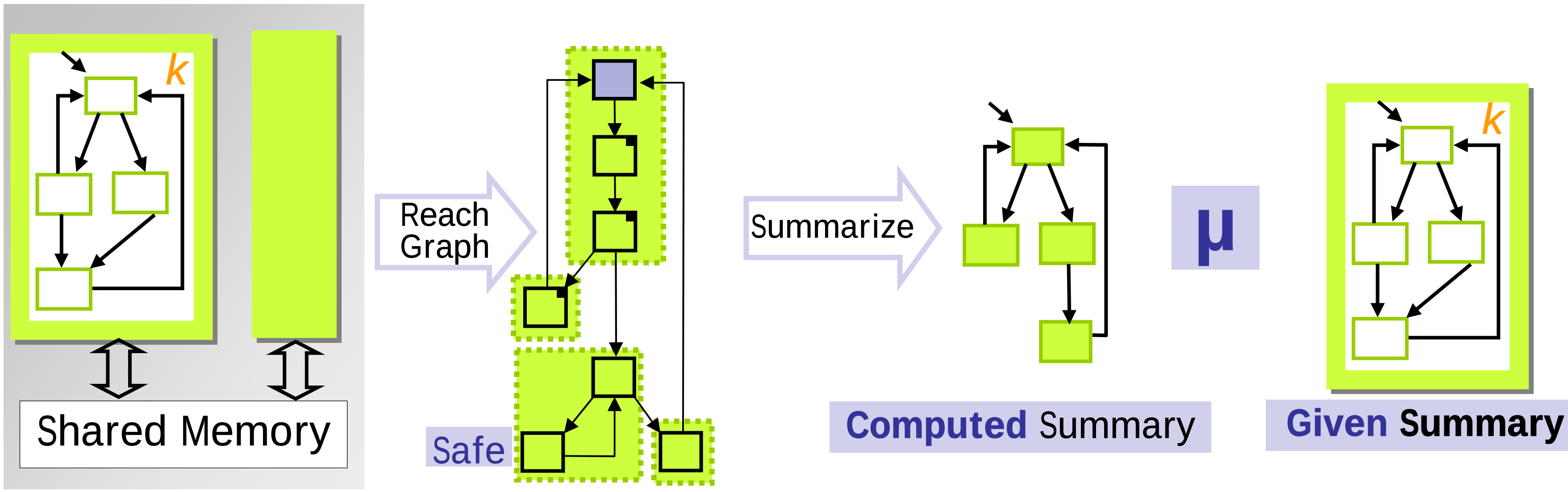
Plan

1. Abstractions against State Explosion
   - Data : Predicate Abstraction
   - **Control : Context Abstraction = Summary + Counting**
2. Thread-Context Reasoning
3. Context Inference
4. Experiments
5. Related Work

Plan

1. Abstractions against State Explosion
   - Data : Predicate Abstraction
   - Control : Context Abstraction = Summary + Counting
2. **Thread-Context Reasoning**
3. Context Inference
4. Experiments
5. Related Work

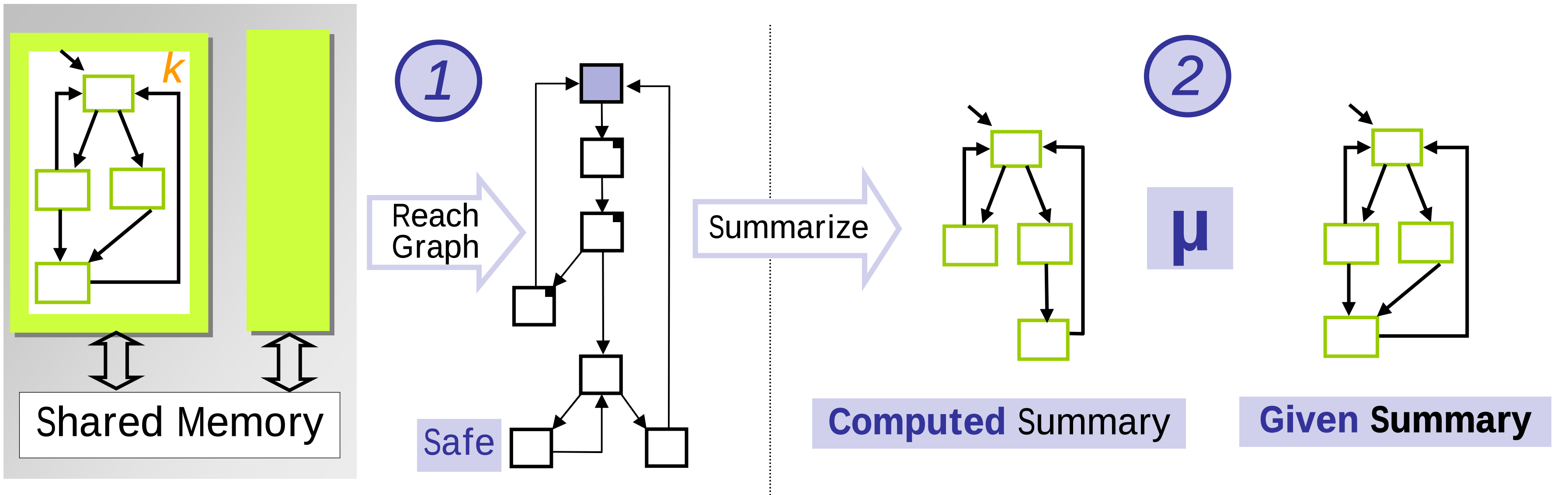
Thread-Context Reasoning



② **Verify** Context Sound

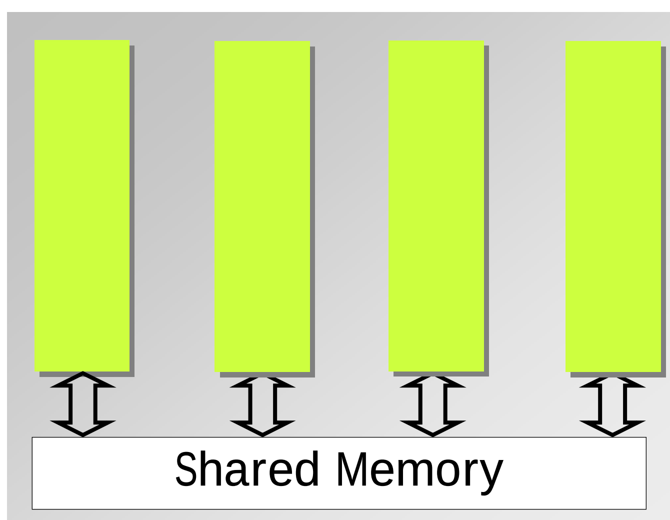
Check Summary Overapproximates single Thread's behavior

Thread-Context Reasoning



Assume-Guarantee

(Use) (Verify)



Safe

Thread-Context Reasoning

Given an Abstraction:
1. Data: *Predicates* 2. Control: *Summary, k*

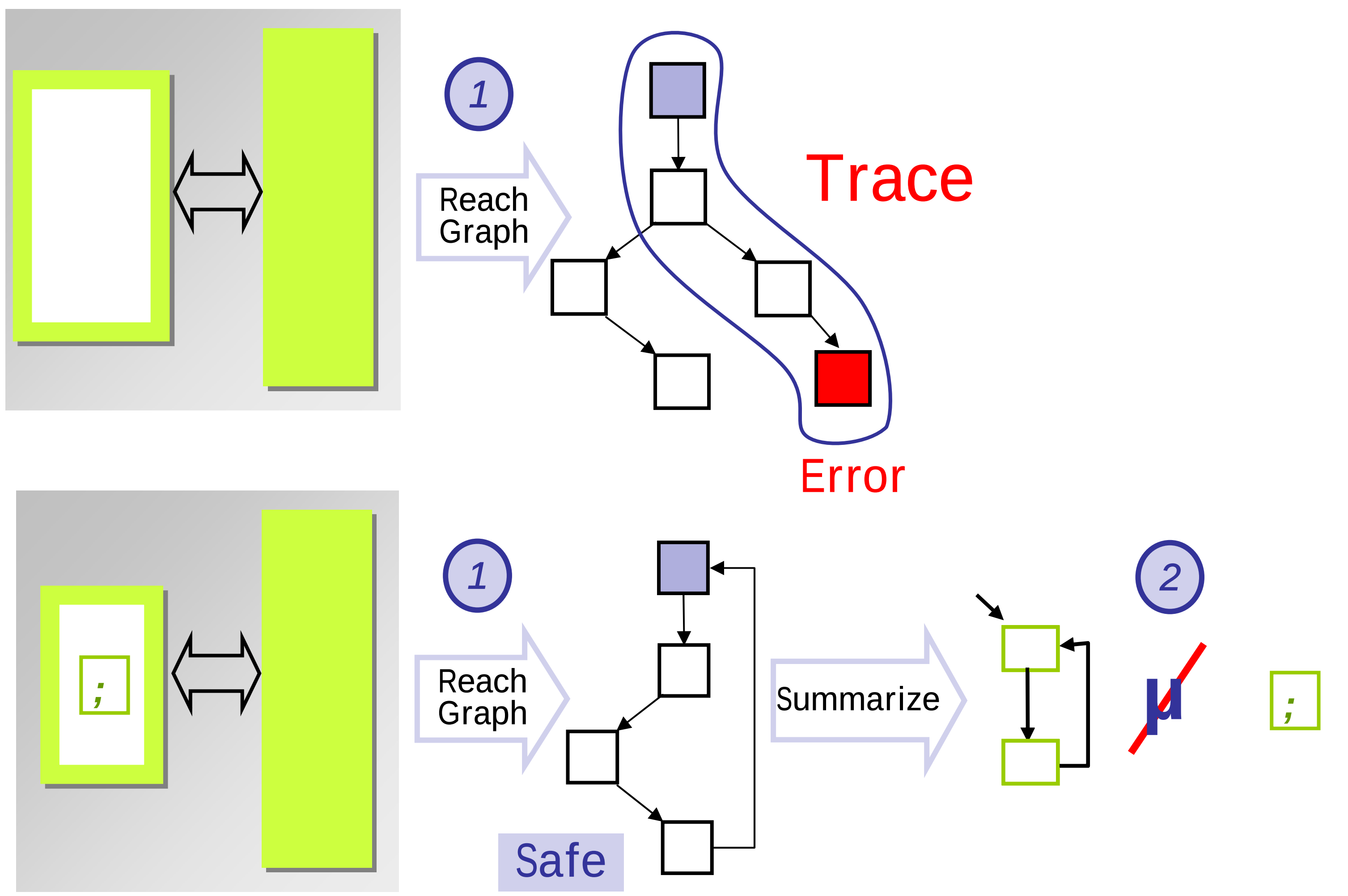
Q: How to find ***predicates*, *summary*, *k* ?**

Plan

1. Abstractions against State Explosion
   - Data : Predicate Abstraction
   - Control : Context Abstraction = Summary + Counting
2. Thread-Context Reasoning
3. **Context Inference**
4. Experiments
5. Related Work

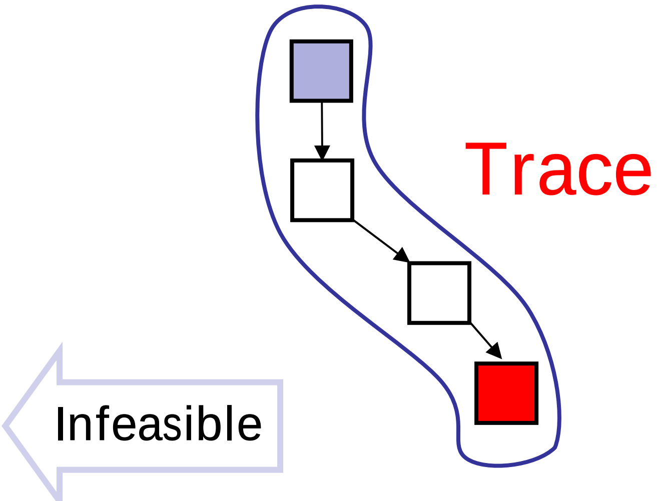
Inference: Build Summary

Abstraction
Preds: P_0 **Ctr:** k_0



Inference: Trace Analysis

Abstraction
Preds: P_1 **Ctr:** k_1



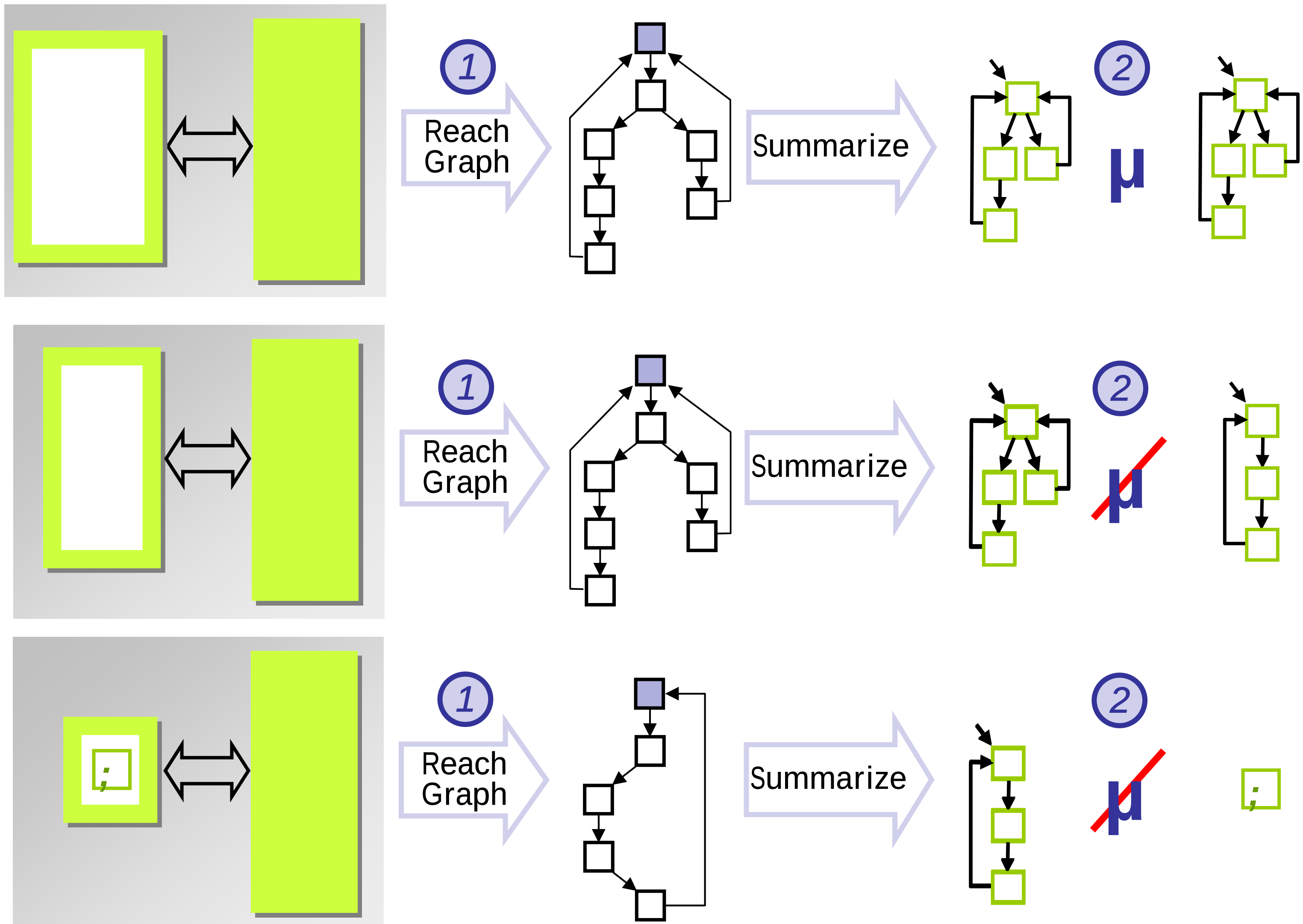
Refine using Trace

Either:

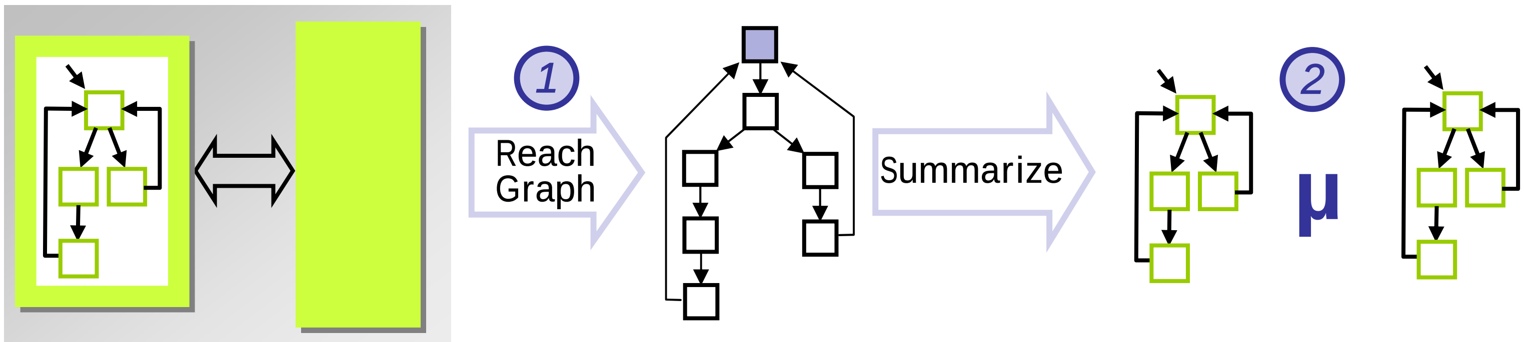
1. Add **new predicates**
2. **Increase** k

Inference: Build Summary

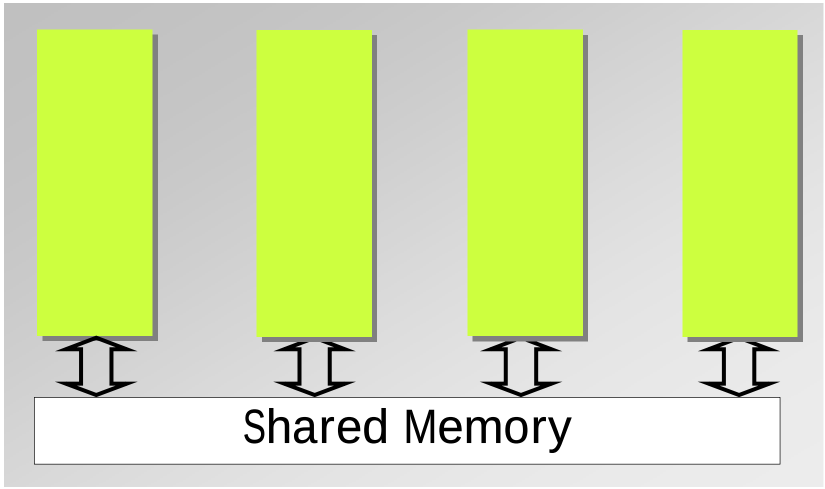
Abstraction
Preds: P_1 **Ctr:** k_1



Context Inferred

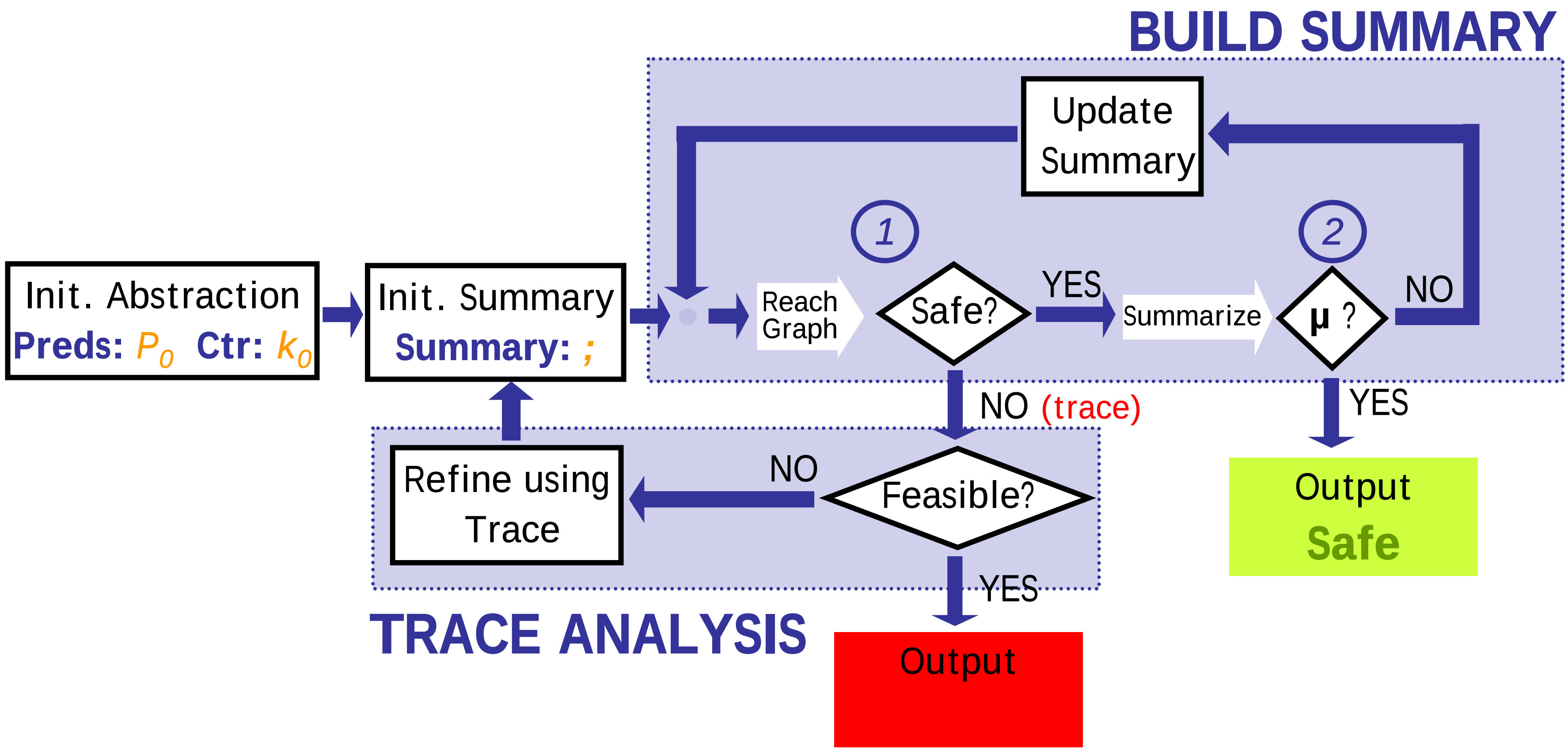


Assume-Guarantee



Safe

Context Inference

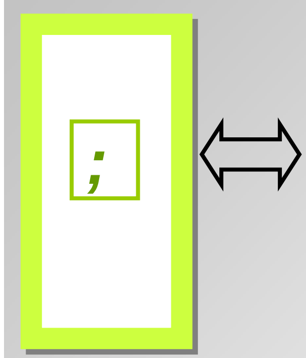


Plan

1. Abstractions against State Explosion
   - Data : Predicate Abstraction
   - Control : Context Abstraction = Summary + Counting
2. Thread-Context Reasoning
3. **Context Inference**
   - **Example**
4. Experiments
5. Related Work

Ex: Races on x

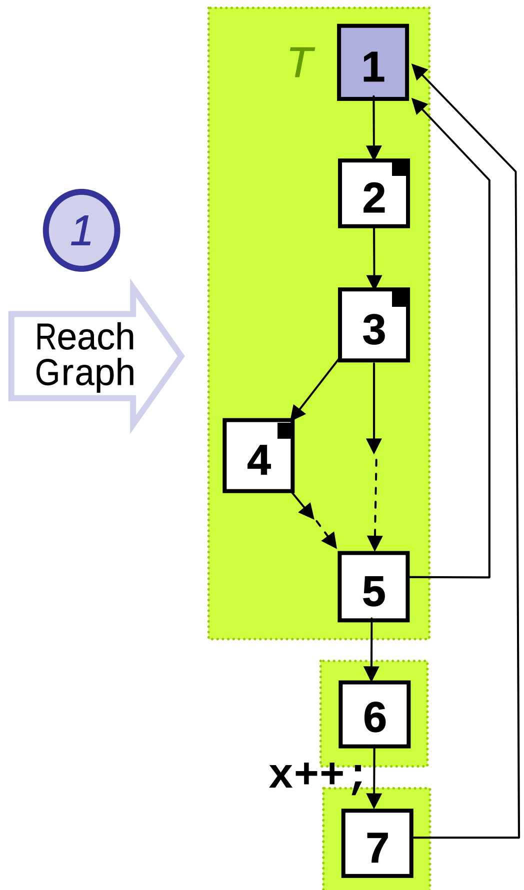
Build Summary

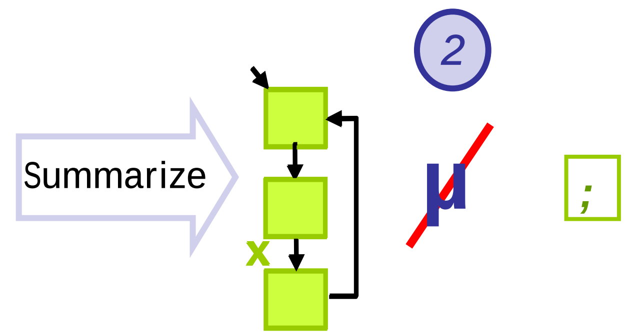



```
1:  while(1){
      atomic{
2:     old := s;
3:     if(s==0){
4:       s := 1;
       }
      }

  // do_work()
      M

5:    if(old==0){
6:      x++;
7:      s:=0;}
}
```



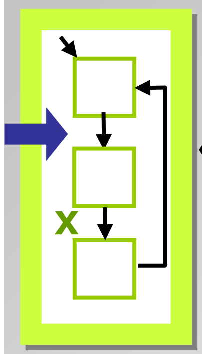


Control-Flow Graph

Abstraction
Preds =;
k=1

Ex: Races on x

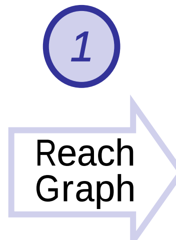
Build Summary

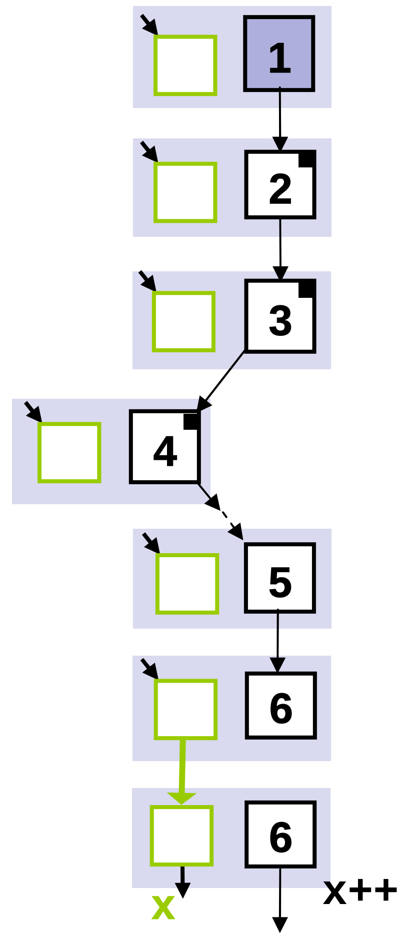



```
1:  while(1){
      atomic{
2:      old := s;
3:      if(s==0){
4:        s := 1;
        }
      }

    // do_work()
      M

5:    if(old==0){
6:      x++;
7:      s:=0;}
    }
```



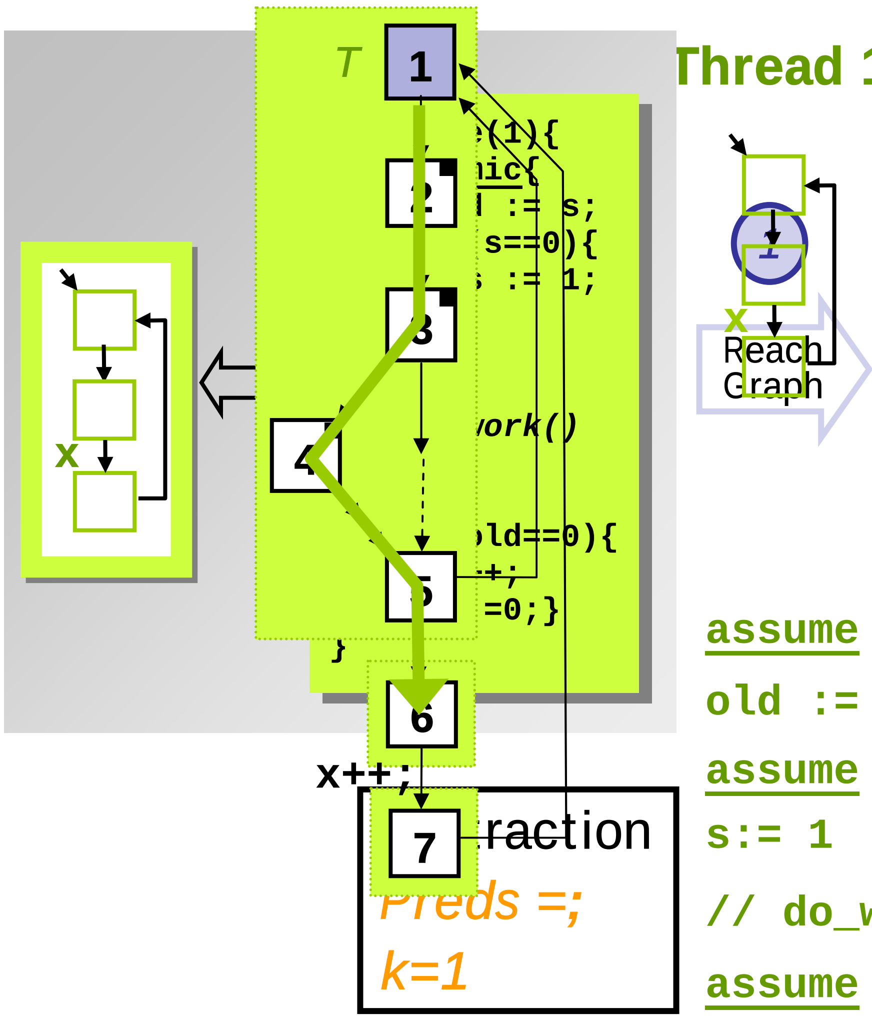


Race

Abstraction
Preds =;
k=1

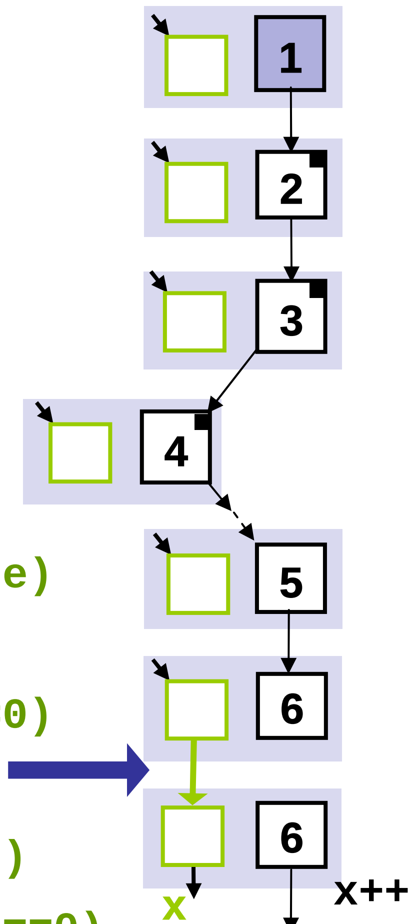
Ex: Races on x

Trace Analysis




```
assume (True)
old := s
assume (s==0)
s:= 1
// do_work()
assume (old==0)
//write x enabled
```

Trace



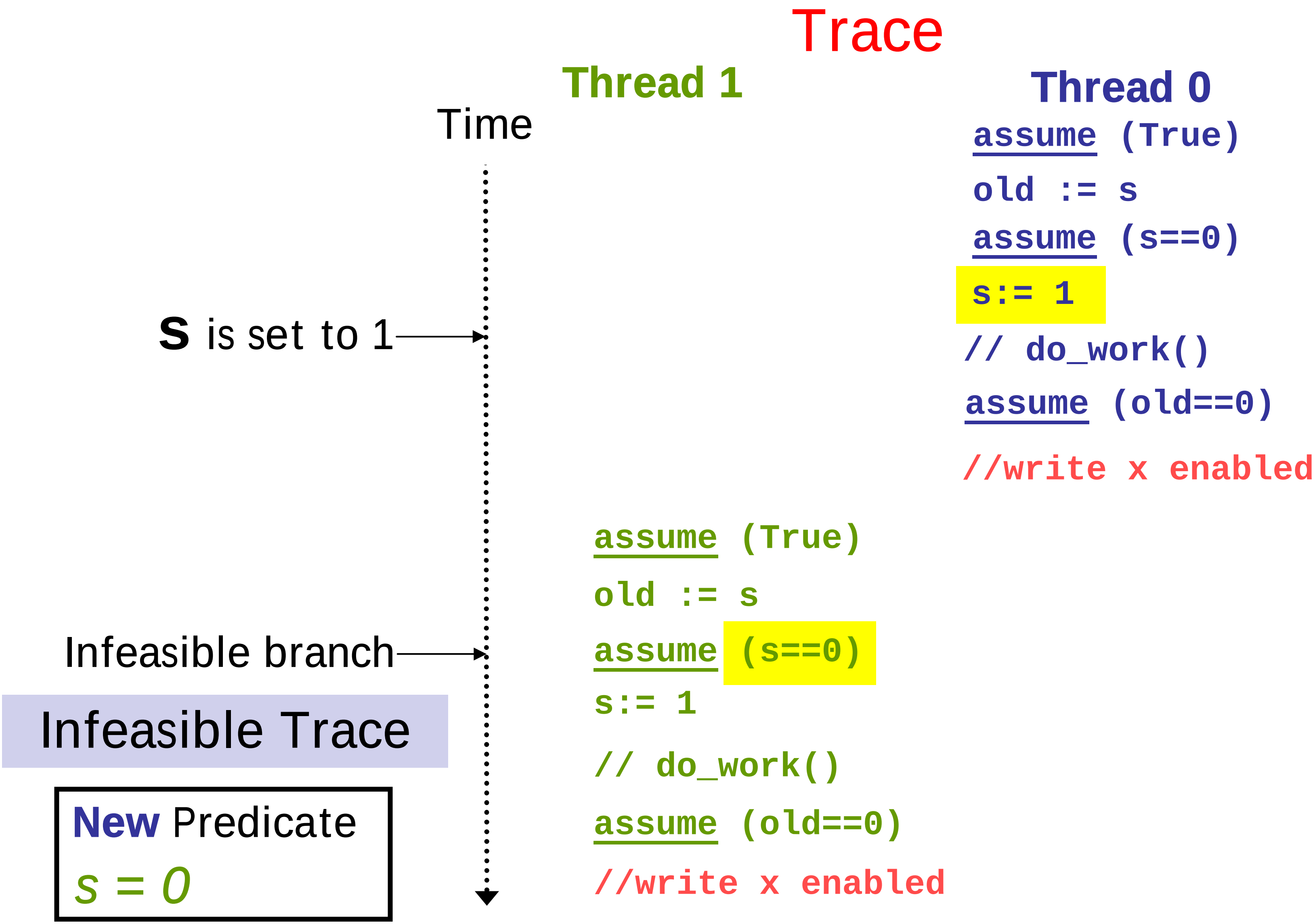
Thread 0

```
assume (True)
old := s
assume (s==0)
s:= 1
// do_work()
assume (old==0)
//write x enabled
```

```
1: while(1){
     atomic{
       old := s;
3:     if(s==0){
4:       s := 1;
       }
     }
     // do_work()
5:   if(old==0){
6:     x++;
7:     s:=0;}
   }
```

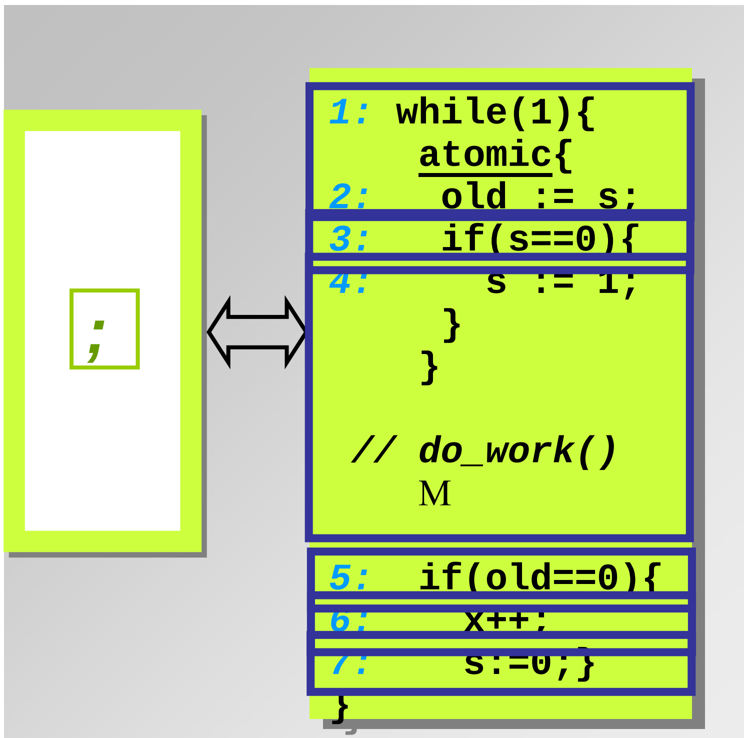
Ex: Races on x

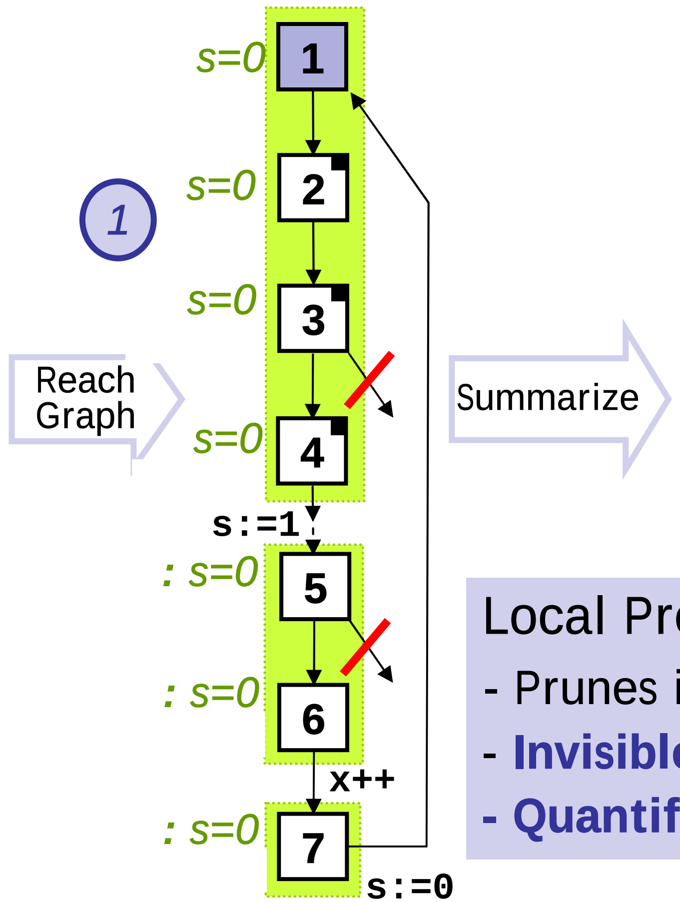
Trace Analysis

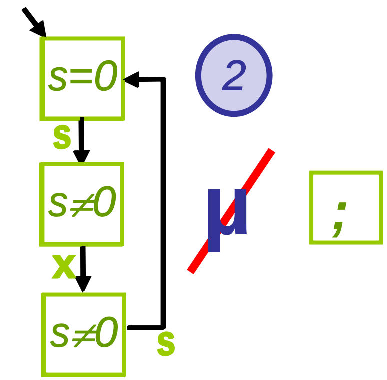


Ex: Races on x

Build Summary







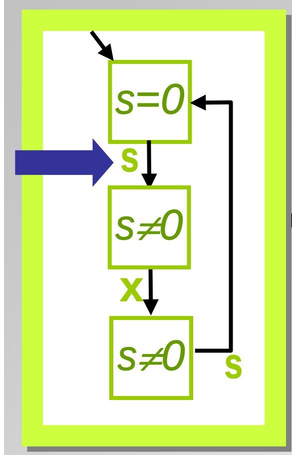
Abstraction
Preds: s=0, old=0
k=1

Local Pred. *old=0*

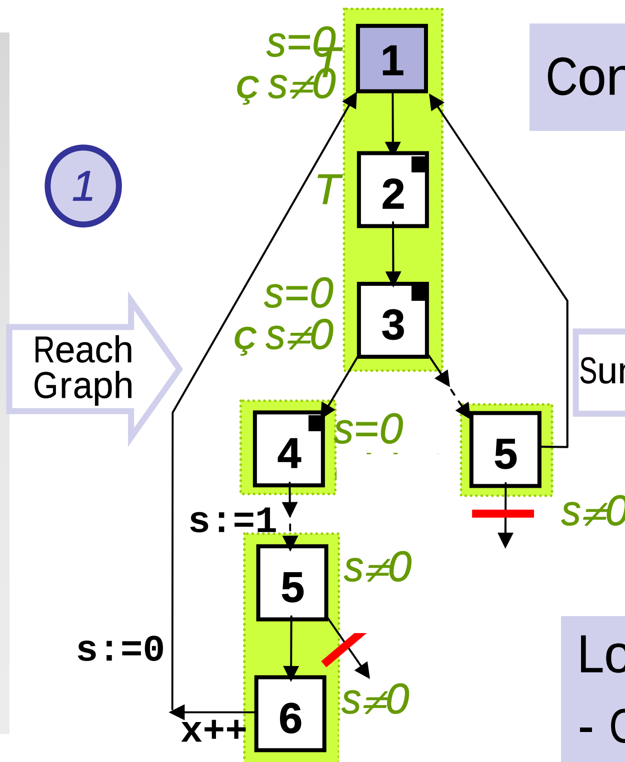
- Prunes infeasible paths
- **Invisible** to other threads
- **Quantified away**

Ex: Races on x

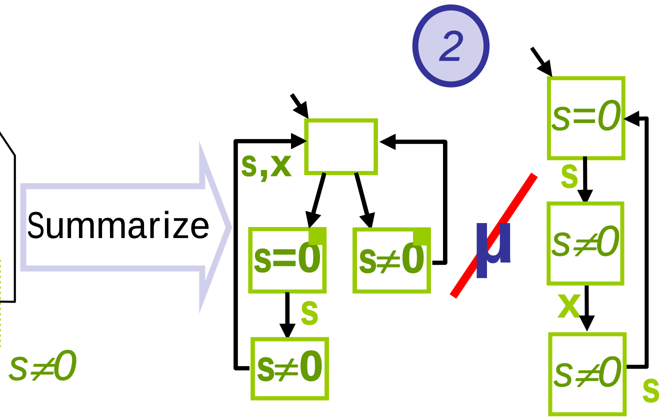
Build Summary




```
1:while(1){
   atomic{
2:   old := s;
3:   if(s==0){
4:      s := 1;
     }
   }
 // do_work()
   M
5: if(old==0){
6:    x++;
7:    s:=0;}
}
```



Context changes **s**



Local Pred. *old=0*

- Cuts locally infeasible paths
- Invisible to other threads
- **Quantified away**

Abstraction

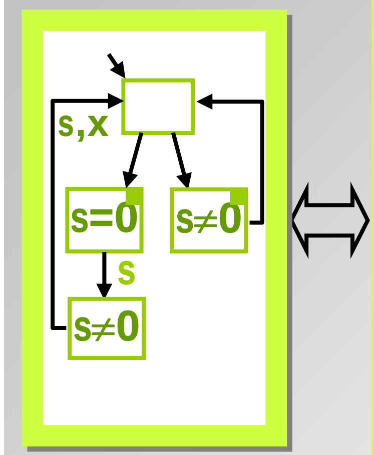
Preds: s=0, old=0

k=1

Ex: Races on x

SAFE

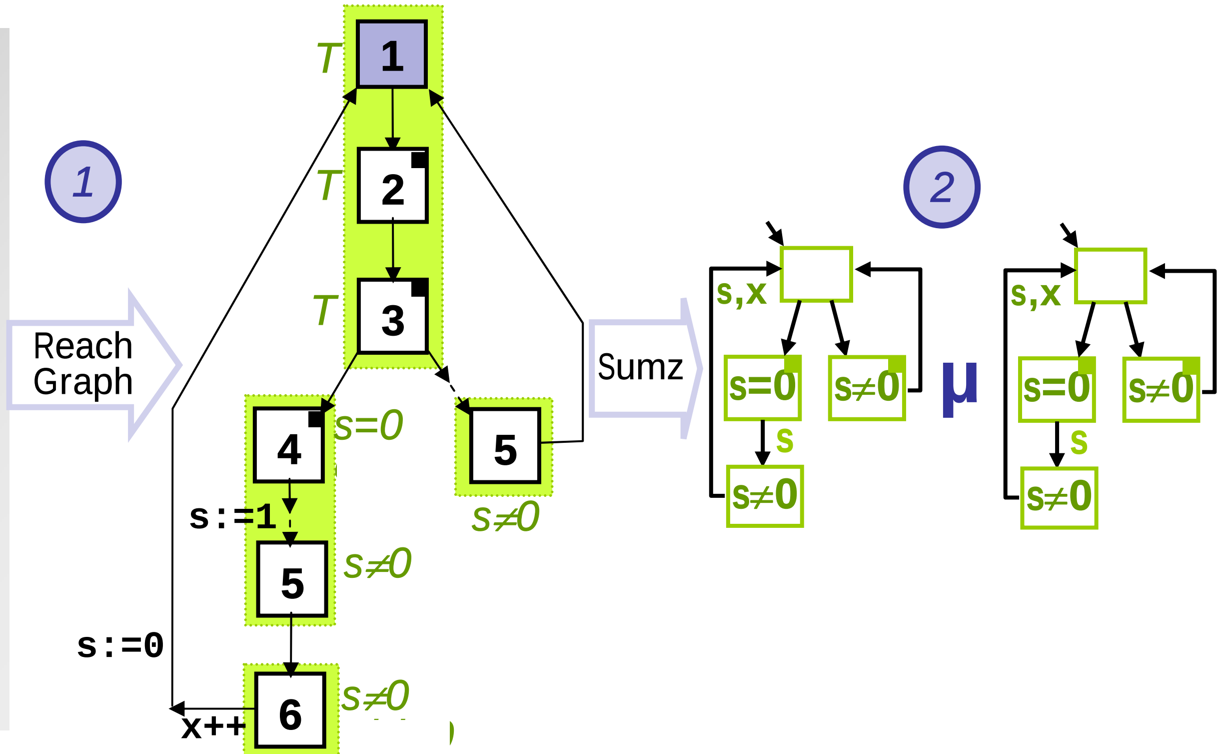
No Races on x




```
1:while(1){
   atomic{
2:   old := s;
3:   if(s==0){
4:      s := 1;
     }
   }

// do_work()
   M

5:  if(old==0){
6:     x++;
7:     s:=0;}
}
```



Abstraction
Preds: s=0, old=0
k=1

Plan

1. Abstractions against State Explosion
   - Data : Predicate Abstraction
   - Control : Context Abstraction = Summary + Counting
2. Thread-Context Reasoning
3. Context Inference
   - Example
4. **Experiments: Races in NesC Programs**
5. Related Work

Race Checking in NesC Programs

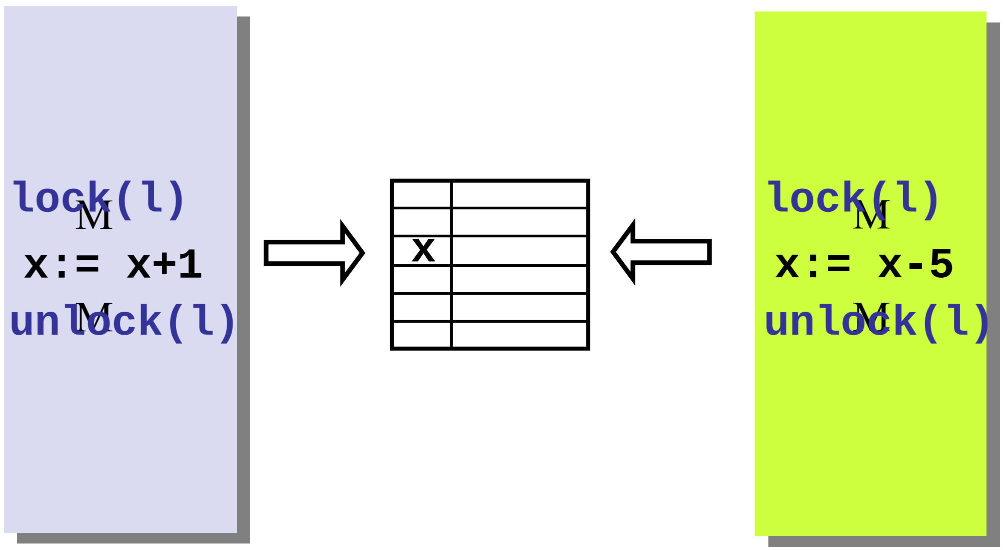
PL for Networked Embedded Systems [Gay et al. 03]

- **Interrupts** fire **events**, which fire other events or post **tasks** which run asynchronously

- Race-freedom important
  - Flow-based analysis
  - Non-trivial synchronization idioms

- Compiled to C

Lock-based Synchronization



A data race on **x** is a state where:

- **Two** threads can access **x**
- One of the accesses is a **write**

Unpredictable, undesirable

Synchronization: Must hold **lock** when accessing x

NesC Synchronization Idioms

```
atomic{
 old:= state;
 if(state==0){
    state:=1;
 }
}
M
if(old==0){
  x++;
  state:=0;
}
```

State-based

Case Study: *sense.nc*

```
atomic{
 old:= state;
 if(state==0){
    state:=1;
 }
}
M
if(old==0){
  x++;
      M
```

Interrupt 1 fires

M

old := state

if (state ==0){

state := 1

M

assume (old == 0){

about to write x

Interrupt 2 fires

M

state := 0

Interrupt 1 fires

M

old := state

assume (state ==0){

state := 1

M

If (old == 0){

about to write x

Interrupt 1 handler **disables** interrupt 2

BLAST finds information proves no races

NesC Race Checking Results

*Program (size)**	*Variable*	*Preds*	*Summary*	*Time*
SecureTosBase (9539 lines)	gTxState	11	23	7m38s
	gTxByteCnt	4	13	1m41s
	gTxRunCrc	4	13	1m50s
	gTxProto	0	9	12s
	gRxHeadIdx	8	64	20m50s
	gRxTailIdx	0	5	2s
Surge (9637 lines)	rec_ptr	4	23	1m18s
	gTxByteCnt	4	15	1m34s
	gTxRunCrc	4	15	1m45s
	gTxState	11	35	9m54s
Sense (3019 lines)	tosPort	6	26	16m25s

* Pre-processed

Limitations

- Predicates
  - How complex are the required invariants ?
  - Are quantified invariants (predicates) needed ?
  - Complex data structures

- Concurrent Interaction Depth
  - Iterations to fixpoint = Interaction depth ...

- State Explosion
  - Counter Abstraction blows up for large counter values k > 3 ...
  - Possible states = P £ k^n

Related Work : Locks

```
lock(l)
 M
x:= x+1
 M
unlock(l)
```

Dynamic LockSet	[Dinning-Schonberg 90] [Savage et al. 97] [Cheng et al. 98] [Choi et al. 02]
Type-based	[Flanagan-Freund 00] [Bacon et al. 00] [Boyapati et al. 02]
Static LockSet	[Sterling 93], [Engler-Ashcraft 03]
Object Usage Graph	[von Praun-Gross 03]

1. **Infer** some lock(s) that protect **x**
2. **Check** lock(s) held when accessing **x**
3. **Report error** if lock(s) not held

Scalable

Restricted to locking

Related Work : State Exploration

State Exploration
(Model Checking)

[Lubachevsky 83]
[Godefroid 97]
[Holzmann][Havelund-Visser]
[Dwyer-Hatcliff][Avrunin-Clarke]
[Musuvathi-Dill-Engler 02]
[Yahav 01]

Any Synch. Idiom
Fixed #threads
Manual Abstraction

State Explosion

Related Work : Assume-Guarantee

- Sequential Programs (Procedure Pre/Postconditions)
  [Hoare 71]
- Concurrent Systems
  [Chandy-Misra 81][]
  [Abadi-Lamport 93]
- Hardware Verification
  [Alur-Henzinger 96][McMillan 97]
  [Eiriksson, McMillan, Henzinger-Qadeer-Rajamani]
- Software Verification
  [Jones 83] [Flanagan-Qadeer-Freund-Seshia 02]
- Automating via Machine Learning
  [Giannapokopolou-Pasareanu-Barringer 03][Alur et.al. 05]

To sum up ...

- Multithreaded programs are hard to verify
  - Data Explosion from many possible values of variables
  - Control Explosion from thread interleaving
- Data Abstraction
  - Track important relationships via predicates
- Control Abstraction
  - Thread Summary = Abstract state machine
  - Context = Summaries + Counting
- Thread-Context Reasoning
  - System = Main Thread + Context
- Iterative Context Inference

Thank You!

http://www.eecs.berkeley.edu/~blast